建模调参

上来先是介绍了几个模型，我看了一下链接挑了重点记录

线性回归最普通的形式是

$$f(x) = w'x + b$$

其中x向量代表一条样本{x1,x2,x3....xn}，其中x1，x2，x3代表样本的各个特征，w是一条向量代表了每个特征所占的权重，b是一个标量代表特征都为0时的预测值，可以视为模型的basis或者bias。看起来很简单的。

这里的w乘以x在线性代数中其实代表的是两个向量的内积，假设w和x均为列向量，即代表了w和x向量的内积w'x。同样的这里的x也可以是一个矩阵X，w与X也可以写成w'X，但是b也要相应的写为向量的形式。

```
#w是列向量 矩阵由一个个列向量构成 y = dot(w_t,X)+b
import numpy as np
w_t,b = np.array([1,2,3,4,5]),1
X = np.array([[1,1,1,1,1],[1,2,5,3,4],[5,5,5,5,5]]).T
y_hat = np.dot(w,X) + b
```

损失函数

线性回归的模型就是这么简单，困难的地方在于我们如何获得w和b这两个向量，在李航老师的统计学习方法中把一个学习过程分为了三部分，模型、策略、算法，为了获得w和b我们需要制定一定的策略，而这个策略在机器学习的领域中，往往描述为真实值与回归值的偏差。

$$loss = (f(x) - y)^2$$

我们希望的是能够减少在测试集上的预测值f(x)与真实值y的差别，从而获得一个最佳的权重参数，因此这里采用最小二乘估计。

这里复习一下之前《数学估计方法》中留下的问题，最小方差无偏估计和最小二乘估计是不一样的，那么他们的区别在哪？最直观的地方是这里表示的是估计点与真实点的差别，而**最小方差无偏估计表示的是估计点与真实数据期望的差别**。

最小二乘

最小二乘优化的思路是线性代数中的矩阵求导，学过导数的人都知道，如果我们想要让loss取到最小，只需要对这个式子进行求导，导数为0的地方就是极值点，也就是使loss最大或者最小的点（实际上是最小的点，因为loss一般是一个往下突的函数，w无限大的时候随便带进去一个值估计出来的值loss都很大，其实求2阶导数也可以看出来）。

任务变成了求这个 $\frac{d((w'X+b)-y)^2}{dw}=0$ 的数学问题。

这里可以参考《矩阵求导公式》里的求导方法，总之求出来是 $2(w'X-y)X^T=0$ ，然后求出来 $w'XX^T=yX^T$ ，这里如果XX^T可逆的话，可以变成 $w'=yX^T(XX^T)^{-1}$ ，这里矩阵求导公式我也记不住，但是凭感觉应该是对的吧。

```
w_t = np.dot(np.dot(y,X.T), np.linalg.inv(np.dot(X,X.T)))
```

1. CART回归树

GBDT是一个集成模型，可以看做是很多个基模型的线性相加，其中的基模型就是CART回归树。

CART树是一个决策树模型，与普通的ID3，C4.5相比，CART树的主要特征是，他是一颗二分树，每个节点特征取值为“是”和“不是”。举个例子，在ID3中如果天气是一个特征，那么基于此的节点特征取值为“晴天”、“阴天”、“雨天”，而CART树中就是“不是晴天”与“是晴天”。

这样的决策树递归的划分每个特征，并且在输入空间的每个划分单元中确定唯一的输出。

1.1 回归树生成

输入：训练数据集D={(x1,y1),(x2,y2),....,(xn,yn)}

输出：一颗回归树 $f(x)=\sum_{m=1}^{M}\hat{c}_m I(x\in R_m)$

一个回归树对应的输入空间的一个划分，假设已经将输入空间划分为M个单元R1,R2,R3....Rm，并且每个单元都有固定的输出值cm，其中I为判别函数。

2. GBDT模型

GBDT模型是一个集成模型，是很多CART树的线性相加。

GBDT模型可以表示为以下形式，我们约定ft(x)表示第t轮的模型，ht(x)表示第t颗决策树，模型定义如下：

$$f_t(x) = \sum_{t=1}^{T} h_t(x)$$

提升树采用前向分步算法。第t步的模型由第t-1步的模型形成，可以写成：

$$f_t(x) = f_{t-1}(x) + h_t(x)$$

损失函数自然定义为这样的：

$$L(f_t(x), y) = L(f_{t-1} + h_t(x), y)$$

虽然整体思路都挺清晰的，但是怎么确定第t步该加上一颗什么样的树确是个大问题。针对这个问题， Freidman提出了用**损失函数的负梯度来拟合本轮损失的近似值，进而拟合一个CART回归树**。即每次需要拟合的是模型的负梯度。第t轮的第i个样本的损失函数的负梯度表示为：

$$r_{t,i} = -[\frac{\delta L(y, f(x_i))}{\delta f(x_i)}]$$

那像你提到有用到Xgboost，你有测试过Xgboost和GBDT的效果区别吗？你认为在你的项目上是什么导致原因导致了这个区别”

“是的，我们有试过，Xgboost的精度要比GBDT高而且效率也要更高。我认为精度高的最大原因是大部分的CTR特征中，我们会将一些稀疏的离散特征转化为连续特征，会产生很多含有缺失值的稀疏列，导致原始GBDT算法效果不好。而Xgboost会对缺失值做一个特殊的处理。在单机的效率高是因为建树时采用了基于分位数的分割点估计算法”

“对缺失值是怎么处理的？”

“在普通的GBDT策略中，对于缺失值的方法是先手动对缺失值进行填充，然后当做有值的特征进行处理，但是这样人工填充会影响数据分布，而且没有什么理论依据。而Xgboost采取的策略是先不处理那些值缺失的样本，先依据有值的特征计算特征的分割点，然后在遍历每个分割点的时候，尝试将缺失样本划入左子树和右子树，选择使损失最优的情况。”

“那你知道在什么情况下应该使用Onehot呢？”

“对于无序特征来说需要做onehot，实践中发现在线性模型中将连续值特征离散化成0/1型特征效果会更好(线性模型拟合连续特征能力弱，需要将连续特征离

散化 成one hot形式提升模型的拟合能力)。所以对于稠密的类别型特征，可以对离散特征做一个OneHot变化，对于稀疏的离散特征采用onehot会导致维度爆炸以及变换后更加稀疏的特征，最好还是采用bin count或者embedding的方法去处理”

“你能讲一下Xgboost和GBDT的区别吗？”

“Xgboost是GBDT算法的一种很好的工程实现，并且在算法上做了一些优化，主要的优化在一下几点。首先Xgboost采用二阶泰勒展开拟合损失函数，可以加速模型收敛；然后加了一个衰减因子，相当于一个学习率，可以减少加进来的树对于原模型的影响，让树的数量变得更多；其次是在原GBDT模型上加了个正则项，对于树的叶子节点的权重做了一个约束；还有增加了在随机森林上常用的col subsample的策略；然后最大的地方在于不需要遍历所有可能的分裂点了，它提出了一种估计分裂点的算法。在工程上做了一个算法的并发实现，具体我并不了解如何实现的”

LGB选择梯度大的样本来计算信息增益。原文给出的理由如下：

> if an instance is associated with a small gradient, the training error for this instance is small and it is already well trained.

如果一个样本的梯度较小，证明这个样本训练的误差已经很小了，所以不需要计算了。我们在XGB的那篇文章中说过，GBDT的梯度算出来实际上就是残差，梯度小残差就小，所以该样本拟合较好，不需要去拟合他们了。

这听起来仿佛很有道理，但问题是丢掉他们会改变数据的分布，还是无法避免信息损失，进而导致精度下降，所以LGB提出了一个很朴实无华且枯燥的方法进行优化。

LGB的优化方法是，在保留大梯度样本的同时，随机地保留一些小梯度样本，同时放大了小梯度样本带来的信息增益。

这样说起来比较抽象，我们过一遍流程： 首先把样本按照梯度排序，选出梯度最大的a%个样本，然后在剩下小梯度数据中随机选取b%个样本，在计算信息增益的时候，将选出来b%个小梯度样本的信息增益扩大 1 - a / b 倍。这样就会避免对于数据分布的改变。

这给我的感觉就是一个公寓里本来住了十个人，感觉太挤了，赶走了六个人，但剩下的四个人要分摊他们六个人的房租。

除了通过部分样本计算信息增益以外，LGB还内置了特征降维技术，思想就是合并那些冲突小的稀疏特征。

举个例子，对于一列特征[1,nan,1,nan,1]和一列特征[nan,1,nan,1,nan]，他们正好可以合并成一列特征[1,2,1,2,1]。LGB的目标就是在于找到这样的特征并且将他们合并在一起。

如果把特征抽象成图中的点，特征之间的冲突看作是图中的边，那么问题就转换为找出图中的社团并使图中的社团数量最少。LGB里提出了一个贪心的策略，按照有权度来为图中所有的点排序，然后把特征合并到度小于某个阈值的社团中或单独创建一个社团。

对于特征如何合并，一个重要的原则就是使合并的两个特征可以被顺利区分出来，LGB采取了一个更改阈值的方法。例如对于特征x∈(0, 10)， 特征y∈(0, 20)，就可以把特征y转换为y∈(10,30)，然后再去合并x与y。

但是更多的情况，对于有数据科学经验的工程师，在使用前往往会避免在特征中掺杂大量稀疏属性。

reduce_mem_usage 函数通过调整数据类型，帮助我们减少数据在内存中占用的空间

```
def reduce_mem_usage(df):
 """ iterate through all the columns of a dataframe and modify the data type
 to reduce memory usage.
 """
 start_mem = df.memory_usage().sum()
 print('Memory usage of dataframe is {:.2f} MB'.format(start_mem))

 for col in df.columns:
 col_type = df[col].dtype

  if col_type != object:
  c_min = df[col].min()
  c_max = df[col].max()
  if str(col_type)[:3] == 'int':
  if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
  df[col] = df[col].astype(np.int8)
  elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
  df[col] = df[col].astype(np.int16)
  elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
  df[col] = df[col].astype(np.int32)
```

```
    elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
    df[col] = df[col].astype(np.int64)
    else:
    if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:
    df[col] = df[col].astype(np.float16)
    elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max:
    df[col] = df[col].astype(np.float32)
    else:
    df[col] = df[col].astype(np.float64)
    else:
    df[col] = df[col].astype('category')

    end_mem = df.memory_usage().sum()
    print('Memory usage after optimization is: {:.2f} MB'.format(end_mem))
    print('Decreased by {:.1f}%'.format(100 * (start_mem - end_mem) / start_mem))
    return df
```

这段真不错啊，可以有效地减少运行时间，以前跑一次太慢了

```
sample_feature = reduce_mem_usage(pd.read_csv('data_for_tree.csv'))
```

```
Memory usage of dataframe is 62099672.00 MB

Memory usage after optimization is: 16520303.00 MB

Decreased by 73.4%
```

效果真的太强了

查看训练的线性回归模型的截距（intercept）与权重(coef)

```
'intercept:'+ str(model.intercept_)

sorted(dict(zip(continuous_feature_names, model.coef_)).items(), key=lambda x:x[1], reverse=True)
```

```
[9]: [('v_6', 3367064.3416419234),
     , ('v_8', 700675.5609399051),
     , ('v_9', 170630.2772322213),
     , ('v_7', 32322.661932046794),
     , ('v_12', 20473.670796955677),
     , ('v_3', 17868.079541493174),
     , ('v_11', 11474.938996699882),
     , ('v_13', 11261.764560014171),
     , ('v_10', 2683.92009059701),
     , ('gearbox', 881.822503924793),
     , ('fuelType', 363.9042507217258),
```

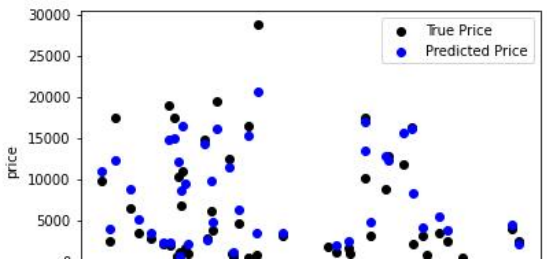
```
from matplotlib import pyplot as plt
```

```
subsample_index = np.random.randint(low=0, high=len(train_y), size=50)
```

绘制特征v_9的值与标签的散点图，图片发现模型的预测结果（蓝色点）与真实标签（黑色点）的分布差异较大，且部分预测值出现了小于0的情况，说明我们的模型存在一些问题

```
plt.scatter(train_X['v_9'][subsample_index], train_y[subsample_index], color='black')
plt.scatter(train_X['v_9'][subsample_index], model.predict(train_X.loc[subsample_index]), color='blue')
plt.xlabel('v_9')
plt.ylabel('price')
plt.legend(['True Price','Predicted Price'],loc='upper right')
print('The predicted price is obvious different from true price')
plt.show()
```

```
The predicted price is obvious different from true price
```

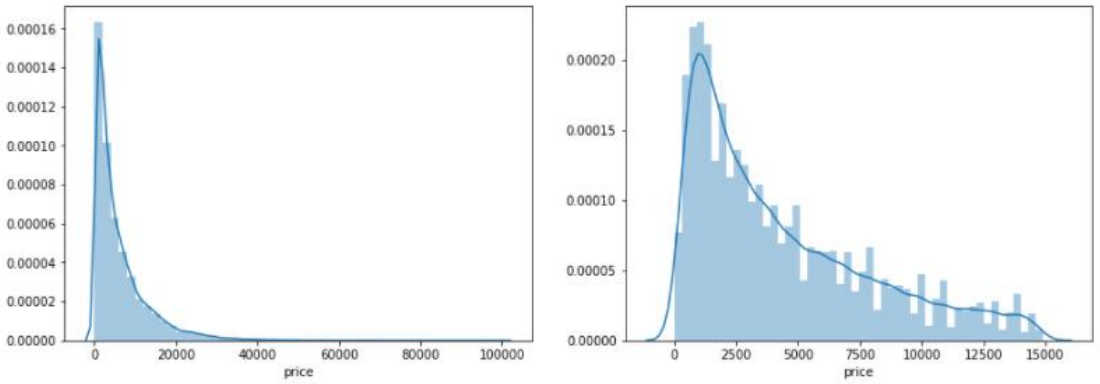


通过作图我们发现数据的标签（price）呈现长尾分布，不利于我们的建模预测。原因是很多模型都假设数据误差项符合正态分布，而长尾分布的数据违背了这一假设。参考博客：https://blog.csdn.net/Noob_daniel/article/details/76087829

```
import seaborn as sns
print('It is clear to see the price shows a typical exponential distribution')
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
sns.distplot(train_y)
plt.subplot(1,2,2)
sns.distplot(train_y[train_y < np.quantile(train_y, 0.9)])
```

```
It is clear to see the price shows a typical exponential distribution
```

[13]: <AxesSubplot:xlabel='price'>



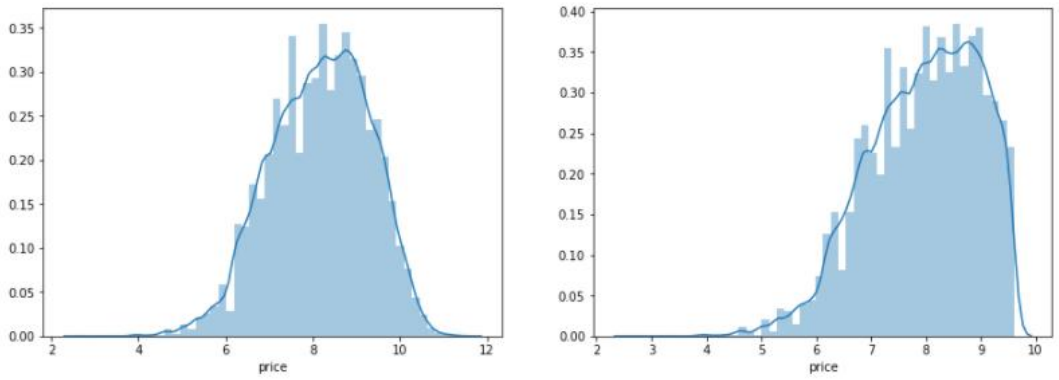
在这里我们对标签进行了 $log(x+1)$ 变换，使标签贴近于正态分布

```
train_y_ln = np.log(train_y + 1)
```

```
import seaborn as sns
print('The transformed price seems like normal distribution')
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
sns.distplot(train_y_ln)
plt.subplot(1,2,2)
sns.distplot(train_y_ln[train_y_ln < np.quantile(train_y_ln, 0.9)])
```

```
The transformed price seems like normal distribution
```

[15]: <AxesSubplot:xlabel='price'>




```
model = model.fit(train_X, train_y_ln)

print('intercept:'+ str(model.intercept_))
sorted(dict(zip(continuous_feature_names, model.coef_)).items(), key=lambda x:x[1], reverse=True)
```

```
intercept:18.75074572712829
```

回归分析的五个基本假设

读了下链接给的文章，很有用

综述

回归分析是一种统计学上分析数据的方法，目的在于了解两个或多个变量间是否相关、相关方向与强度，并建立数学模型。以便通过观察特定变量（自变量），来预测研究者感兴趣的变量（因变量）。

总的来说，回归分析是一种参数化方法，即为了达到分析目的，需要设定一些“自然的”假设。如果目标数据集不满足这些假设，回归分析的结果就会出现偏差。因此想要进行成功的回归分析，我们就必须先证实这些假设。

回归分析的五个基本假设

1. 线性性 & 可加性

假设因变量为Y，自变量为X_1，X_2，则回归分析的默认假设为

$Y=b+a_1X_1+a_2X_2+\varepsilon$。

线性性：X_1每变动一个单位，Y相应变动a_1个单位，与X_1的绝对数值大小无关。

可加性：X_1对Y的影响是独立于其他自变量（如X_2）的。

2. 误差项（ε）之间应相互独立。

若不满足这一特性，我们称模型具有**自相关性**（Autocorrelation）。

3. 自变量（X_1，X_2）之间应相互独立。

若不满足这一特性，我们称模型具有**多重共线性性**（Multicollinearity）。

4. 误差项（ε）的方差应为常数。

若满足这一特性，我们称模型具有**同方差性**（Homoskedasticity），若不满足，则为**异方差性**（Heteroskedasticity）。

5. 误差项（ε）应呈正态分布。

假设失效的影响

1. 线性性 & 可加性

若事实上变量之间的关系不满足线性性（如含有X_{21}, X_{31} 项），或不满足可加性（如含有$X_1 \cdot X_2$项），则模型将无法很好的描述变量之间的关系，极有可能导致很大的**泛化误差**（generalization error）

2. 自相关性（Autocorrelation）

自相关性经常发生于时间序列数据集上，后项会受到前项的影响。当自相关性发生的时候，我们测得的标准差往往会**偏小**，进而会导致置信区间**变窄**。

假设没有自相关性的情况下，自变量X的系数为15.02而标准差为2.08。假设同一样本是有自相关性的，测得的标准差可能会只有1.20，所以置信区间也会从$(12.94,17.10)$缩小到$(13.82,16.22)$。

3. 多重共线性性（Multicollinearity）

如果我们发现本应相互独立的自变量们出现了一定程度（甚至高度）的相关性，那我们就很难得知自变量与因变量之间真正的关系了。

当多重共线性性出现的时候，变量之间的联动关系会导致我们测得的标准差**偏大**，置信区间**变宽**。

采用岭回归，Lasso回归或弹性网（ElasticNet）回归可以一定程度上减少方差，解决多重共线性性问题。因为这些方法，在最小二乘法的基础上，加入了一个与回归系数的模有关的惩罚项，可以收缩模型的系数。

岭回归：$=\operatorname{argmin}_{\beta\in Rp}(\|y-X\beta\|_{22}+\lambda\|\beta\|_{22})$

Lasso回归：$=\operatorname{argmin}_{\beta\in Rp}(\|y-X\beta\|_{22}+\lambda\|\beta\|_{1})$

弹性网回归：$=\operatorname{argmin}_{\beta\in Rp}(\|y-X\beta\|_{22}+\lambda_1\|\beta\|_{1}+\lambda_2\|\beta\|_{22})$

$$where\ \|Z\|_p=(\sum_{i=1N}|Z_i|_p)^{(1/p)}$$

4. 异方差性（Heteroskedasticity）

异方差性的出现意味着误差项的方差不恒定，这常常出现在有异常值（Outlier）的数据集上，如果使用标准的回归模型，这些异常值的重要性往往被高估。在这种情况下，标准差和置信区间不一定会变大还是变小。

5. 误差项（ε）应呈正态分布

如果误差项不呈正态分布，意味着置信区间会变得很不稳定，我们往往需要重点关注一些异常的点（误差较大但出现频率较高），来得到更好的模型。

在这里记录一下关于模型复杂度对拟合以及调整网络节点数的思考：

欠拟合就是训练过程中误差难以下降，过拟合就是训练之后，测试误差要远比训练误差大。

如果模型复杂度太低（参数过少），即模型可训练空间太小，就难以训练出有效的模型，便会出现欠拟合。

如果模型复杂度太高（参数很多），即模型可训练空间很大，在大量样本输入后容易训练过头，便会出现过拟合。

所以控制好模型复杂度（参数数量），是调整欠拟合和过拟合的一种方法。

换句话说，可以通过训练效果的图表判断是过拟合还是欠拟合，以此为依据调整网络的结构。

比如如果欠拟合了，表示无法充分训练，可以将网络层的节点数量调大一些。

这样就找到一种调整网络节点数的依据，而不是仅靠经验。

正则化（Regularization）
机器学习中几乎都可以看到损失函数后面会添加一个额外项，常用的额外项一般有两种，一般英文称作 ℓ 1 \ell_1ℓ
1

-norm 和 ℓ 2 \ell_2ℓ
2

-norm，中文称作 L1正则化 和 L2正则化，或者 L1范数 和 L2范数。

L1正则化和L2正则化可以看做是损失函数的惩罚项。所谓『惩罚』是指对损失函数中的某些参数做一些限制。对于线性回归模型，使用L1正则化的模型建叫做Lasso回归，使用L2正则化的模型叫做Ridge回归（岭回归）。下图是Python中Lasso回归的损失函数，式中加号后面一项α | | w | | 1 \alpha||w||_1α||w||
1

即为L1正则化项。

下图是Python中Ridge回归的损失函数，式中加号后面一项α | | w | | 2 2 \alpha||w||_2^2α||w||
2
2

即为L2正则化项。

一般回归分析中w ww表示特征的系数，从上式可以看到正则化项是对系数做了处理（限制）。L1正则化和L2正则化的说明如下：

L1正则化是指权值向量w ww中各个元素的绝对值之和，通常表示为

$$\min_{w} \frac{1}{2n_{samples}}||Xw-y||_2^2+\alpha||w||_1$$

L2正则化是指权值向量w ww中各个元素的平方和然后再求平方根（可以看到Ridge回归的L2正则化项有平方符号），通常表示为

$$\min_{w} ||Xw-y||_2{}^2+\alpha||w||_2{}^2$$

一般都会在正则化项之前添加一个系数，Python的机器学习包sklearn中用α \alphaα表示，一些文章也用λ \lambdaλ表示。这个系数需要用户指定。

那添加L1和L2正则化有什么用？下面是L1正则化和L2正则化的作用，这些表述可以在很多文章中找到。

L1正则化可以产生稀疏权值矩阵，即产生一个稀疏模型，可以用于特征选择
L2正则化可以防止模型过拟合（overfitting）；一定程度上，L1也可以防止过拟合
稀疏模型与特征选择的关系
上面提到L1正则化有助于生成一个稀疏权值矩阵，进而可以用于特征选择。为什么要生成一个稀疏矩阵？

稀疏矩阵指的是很多元素为0，只有少数元素是非零值的矩阵，即得到的线性回归模型的大部分系数都是0. 通常机器学习中特征数量很多，例如文本处理时，如果将一个词组（term）作为一个特征，那么特征数量会达到上万个（bigram）。在预测或分类时，那么多特征显然难以选择，但是如果代入这些特征得到的模型是一个稀疏模型，表示只有少数特征对这个模型有贡献，绝大部分特征是没有贡献的，或者贡献微小（因为它们前面的系数是0或者是很小的值，即使去掉对模型也没有什么影响），此时我们就可以只关注系数是非零值的特征。这就是稀疏模型与特征选择的关系。

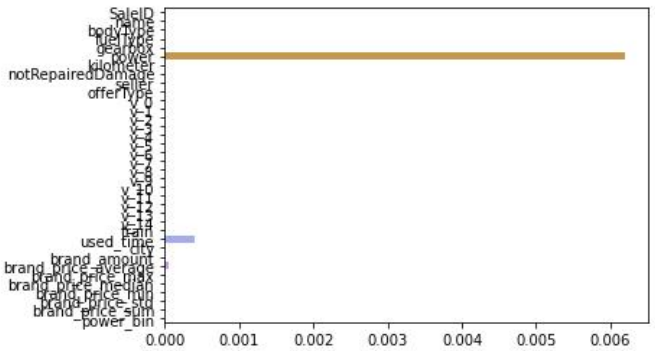
L2正则化在拟合过程中通常都倾向于让权值尽可能小，最后构造一个所有参数都比较小的模型。因为一般认为参数值小的模型比较简单，能适应不同的数据集，也在一定程度上避免了过拟合现象。可以设想一下对于一个线性回归方程，若参数很大，那么只要数据偏移一点点，就会对结果造成很大的影

响；但如果参数足够小，数据偏移得多一点也不会对结果造成什么影响，专业一点的说法是『抗扰动能力强』

L1正则化有助于生成一个稀疏权值矩阵，进而可以用于特征选择。如下图，我们发现power与userd_time特征非常重要。

```
model = Lasso().fit(train_X, train_y_ln)
print('intercept:'+ str(model.intercept_))
sns.barplot(abs(model.coef_), continuous_feature_names)
```

```
intercept:8.67218477236799
[42]: <AxesSubplot:>
```



Grid Search 网格搜索

GridSearchCV：一种调参的方法，当你算法模型效果不是很好时，可以通过该方法来调整参数，通过循环遍历，尝试每一种参数组合，返回最好的得分值的参数组合

比如支持向量机中的参数 C 和 gamma ，当我们不知道哪个参数效果更好时，可以通过该方法来选择参数，我们把C 和gamma 的选择范围定位[0.001,0.01,0.1,1,10,100]

每个参数都能组合在一起，循环过程就像是在网格中遍历，所以叫网格搜索

感觉贝叶斯优化比较方便诶

贝叶斯优化方法

贝叶斯优化通过基于目标函数的过去评估结果建立替代函数（概率模型），来找到最小化目标函数的值。贝叶斯方法与随机或网格搜索的不同之处在于，它在尝试下一组超参数时，会参考之前的评估结果，因此可以省去很多无用功。

超参数的评估代价很大，因为它要求使用待评估的超参数训练一遍模型，而许多深度学习模型动则几个小时几天才能完成训练，并评估模型，因此耗费巨大。贝叶斯调参发使用不断更新的概率模型，通过推断过去的结果来“集中”有希望的超参数。

Python中的选择

Python中有几个贝叶斯优化库，它们目标函数的替代函数不一样。在本文中，我们将使用Hyperopt，它使用Tree Parzen Estimator（TPE）。其他Python库包括Spearmint（高斯过程代理）和SMAC（随机森林回归）。

优化问题的四个部分

贝叶斯优化问题有四个部分：

- 目标函数：我们想要最小化的内容，在这里，目标函数是机器学习模型使用该组超参数在验证集上的损失。
- 域空间：要搜索的超参数的取值范围
- 优化算法：构造替代函数并选择下一个超参数值进行评估的方法。
- 结果历史记录：来自目标函数评估的存储结果，包括超参数和验证集上的损失。

贝叶斯调参的代码似乎和csdn的不太一样，感觉csdn的更加详细一些，附上链接

https://blog.csdn.net/linxid/article/details/81189154

数据挖掘训练营结束了，感觉分享，收获很大